Title
Implementation of the OPU Instruction Set Architecture on the Microsemi Polarfire 300 Field-Programmable Gate Array

Author
Delhez, Louis Jean Eric

Publication Date
2020

Peer reviewed|Thesis/dissertation

UNIVERSITY OF CALIFORNIA

Los Angeles

Implementation of the OPU Instruction Set Architecture on the Microsemi Polarfire 300 Field-Programmable Gate Array

A thesis submitted in partial satisfaction

of the requirements for the degree of Master of Science

in Electrical and Computer Engineering

by

Louis Jean Eric Delhez

2020

ABSTRACT OF THE THESIS

Implementation of the OPU Instruction Set Architecture on the Microsemi Polarfire 300 Field-Programmable Gate Array

by

Louis Jean Eric Delhez

Master of Science in Electrical and Computer Engineering

University of California, Los Angeles, 2020

Professor Lei He, Chair

Deep learning is a fast-growing field with numerous promising applications that, unfortunately, demands large computing power for both training and inference tasks. To meet this demand, numerous hardware accelerators have thus been designed. Currently, however, these platforms are being developed independently from each other, and, as a result, there is a lack of compatibility between them. Notably, there is a need for standardization of the interface between hardware accelerators and software. UCLA's OPU is an ISA that aims at solving this issue. Contrary to general-purpose ISAs, OPU is designed to adequately express the computations involved in deep learning models, which allows for simple compilation and efficient cores.

Prior to this work, only two fully-featured cores implementing the OPU ISA had been designed, both targeted at Xilinx SRAM-based FPGAs. However, flash-based FPGAs can offer several advantages thanks to their different technology. They are more secure, more reliable, and can yield a lower power consumption. All three of these characteristics being potentially highly valuable for deep learning accelerators, especially those embedded in edge devices, a new OPU core is here developed and mapped to a flash-based FPGA. More specifically, the potential of the MPF300 FPGA as a platform for the OPU ISA is evaluated. This represents the first OPU core implemented on an FPGA that is not manufactured by Xilinx. In addition, this design is also the first OPU core capable of operating on floating-point numbers, which simplifies the compilation of models. As such, this work contributes to the diversification of the catalog of available OPU cores, which increases the relevance of this ISA.

While prior work affirms that, on Xilinx FPGAs, 8-bit floating-point arithmetic is more area-efficient than 8-bit integer arithmetic, the opposite is found in this work for Microsemi FPGAs. As a consequence, it is established that the optimum manner to perform large floating-point dot products on the MPF300 is to convert the operands to wider integers, on the device, then complete the computations using integer arithmetic. In contrast to Xilinx FPGAs, 5-bit mantissas are here preferred over 4-bit mantissas. Additionally, due to the lower ratio of the number of LUTs to DSPs of the MPF300, the relative resource utilization is found to be significantly higher here compared to the existing implementations.

This new OPU core is found to be in average 1.7 times more energy-efficient than the existing similarly-sized implementation of the OPU ISA. Furthermore, the new core is in average 2 times faster than the Nvidia Jetson Nano platform, while consuming the same amount of power. These results further prove the relevance of the OPU ISA. In addition, this demonstrates that flash-based FPGAs, too, are a viable option for deep learning acceleration. The scarcity of these FPGAs in the relevant literature is thus not justified. Nevertheless, analysis of the core shows that the layout of modern FPGAs is in general suboptimal for the task of machine learning acceleration. In particular, the placement of the hard resources of the device tends to cause congestion on the device that reduces performance. This suggests the need for the development of specialized FPGAs for this task.

The thesis of Louis Jean Eric Delhez is approved.

Puneet Gupta

Dejan Markovic

Lei He, Committee Chair

University of California, Los Angeles

2020

Contents

List of Figures

List of Tables

Acknowledgments

This work was performed while being a recipient of a Fellowship of the Belgian American Educational Foundation.

Chapter 1

Introduction

Undeniably, machine learning is, for good reasons, a highly popular field of research with countless revolutionizing applications. In particular, artificial neural networks, which vaguely mimic the interconnection of real animal neurons, have been found to be effective at solving various tasks that traditional programming cannot solve. For instance, in [3], such a network is employed to segment pictures of real-life scenes into the different objects that compose them, which is a necessary post-processing task for numerous applications including autonomous driving. In [20], an artificial neural network is trained to predict the true color of pixels in pictures. In that way, the authors are able to re-colorize black and white picture captured more than a century ago. Neural networks can also be used for other artistic purposes such as in [12] where creative paintings that imitate the style of known painters are generated by combining multiple images. Even more remarkably, these models could also be utilized to improve our health. For instance, [15] shows that tumors can be

accurately detected from medical images of patients, automatically, using artificial neural networks. Clearly, those machine learning algorithms have proven themselves to be highly valuable, and one should thus expect to see even more applications of neural networks in the near future.

While neural networks have been shown to be able to solve many challenging problems, they unfortunately tend to be highly compute-hungry. Because of this, their deployment in real applications is often costly, which is a major factor preventing their widespread utilization. To solve this issue, many innovative solutions have been proposed. On the one hand, software approaches, including quantization [14, 13] and pruning [17, 29], have been developed to reduce the overall complexity of these algorithms. On the other hand, novel hardware solutions have been studied to provide acceleration. In particular, graphics processing units (GPUs) have emerged as a popular platform for executing neural network models, notably thanks to the convenience offered by efficient software libraries [7]. However, while these devices are fast and widely available, their power consumption is often relatively high. For this reason, numerous specialized devices, both FPGA-based designs and ASICs, have been proposed such as [1, 6, 23, 10, 5]. All of these devices can provide an considerable computation speedup with excellent power-efficiency. They therefore are a promising solution for the acceleration of artificial neural networks.

Unfortunately, however, these accelerators are generally far less convenient to use in prac-

tice than GPUs. Because of this, deploying these devices in actual products is challenging. Fortunately, most of this issue could be solved by uniforming the software toolchains supporting these devices. In that context, similarly to RISC-V [2] for general purpose computing, OPU [39] is an ISA with the ambitious goal of standardization of specialized neural network accelerator platforms. Until now, only two fully-featured cores implementing this novel ISA have been designed. Experiments on these cores shows that they ca yield promising performance and energy-efficiency, which validates the approach taken by OPU.

The existing OPU cores have both been implemented on SRAM-based FPGAs. Though these cores are themselves already quite power-efficient, flash-based FPGAs has the potential to reduce this power consumption even further thanks to their different technology. Naturally, this would be valuable to extend the scope of OPU to applications with tighter power budgets. In addition to the reduced power consumption, flash-based FPGA provide more security and reliability than SRAM-based FPGAs. This is especially interesting given the number of revolutionizing applications of machine learning that are life-critical, notably in the medical and transport sectors. For all of these reasons, flash-based FPGA have been used in the literature to accelerate some small neural network models such as in [18] and [34]. However, despite their potential advantages, usage of flash-based FPGAs remains uncommon for the acceleration of large modern models. In this context, a new implementation of the OPU ISA is designed in this work. This core is specially optimized for a flash-based FPGA, the Microsemi MPF300, with the goal of creating a more power-efficient platform

than the existing cores, while maintaining relatively high compute performance.

In Chapter 2 of this thesis, the OPU ISA is introduced along with its exiting implementations. Chapter 3 gives a brief review of FPGA technology, while highlighting the differences between the FPGA devices used for the existing two cores and the one considered in this work. Then, Chapter 4 details some design choices of this new OPU core. In particular, the design of the dot product unit is thoroughly studied. Chapter 5 presents the performance and energy-efficiency of this new implementation. Finally, Chapter 6 discusses the current performance bottleneck of the platform, and suggests modifications that could be applied to modern FPGAs in order to make them more adequate for machine learning acceleration.

Chapter 2

OPU

OPU, short for *Open Processing Unit*, is a new ISA created especially for task of deep learning inference acceleration. This ISA was first designed at the Design Automation Laboratory, at UCLA, and introduced in [39].

The rest of this chapter describes the motivation behind the creation of OPU, and the major characteristics of the OPU ISA. Then, an overview of the existing cores implementing this ISA is given.

2.1 Motivation

With the rising popularity of deep learning algorithms, the need for appropriate acceleration is rapidly increasing as well. To meet this demand, several companies have developed their

own hardware solutions, including Google [23] and Amazon [1]. However, currently, the process of deploying deep learning models is still often complicated task and is performed differently depending on the type of model, the high-level language used to describe it, and the targeted platform. Each such platform also employs different sets of software tools that each needs to be developed, maintained, and kept up-to-date. As a result, the current approach is thus overall not ideal, and slows the larger adoption of deep learning accelerators.

In this context, an ISA that standardizes the interface between hardware accelerators and software would be highly valuable. Existing general-purpose ISAs and vector processor ISAs, however, are not ideal to fulfill this role as those define an unnatural hardware/software interface when considering deep learning models. To solve this issue, the OPU ISA has been designed specially to elegantly express the computations of modern neural networks. Thanks to this property, efficient compiler toolchains can be easily implemented to convert the high-level description of models to executable OPU binaries. Likewise, OPU has been designed in a manner that allows for efficient hardware implementations.

With OPU, the same unique software suite could be used for a large variety of accelerators regardless of their brand, size, whether they are employed in the cloud or on the edge, or whether they are implemented on FPGA or not. As such, OPU could help simplify the deployment of deep learning models and thereby support the widespread adoption of AI in consumer applications. To serve this goal, OPU is also planned to be released as a free and

open-source ISA, with the hope that it would progressively be adopted by the academia and the industry. In addition to the obvious cost savings, this open-source ISA would encourage the design of numerous accelerators optimized for different end-applications, depending on their specific requirements in terms of performance, power consumption, cost, security, or reliability. Also, an open-source machine learning ISA would promote the extension of its instruction set to meet the custom needs of each particular application.

2.2 Overview of the OPU ISA

Just like RISC-V, OPU is intended to be a flexible ISA that can be used in a board range of applications, from edge computing to cloud computing. For this reason, the OPU ISA is characterized by a few parameters. In that regard, OPU is thus not a unique ISA but rather a family of ISAs. Of these parameters, the most important is the integer `NUM_MAC`, which indicates the number of multiply-accumulate operation that can performed in parallel by the architecture. A larger value of `NUM_MAC` will thus generally correspond to a faster but more power-hungry accelerator.

In addition to this parametrization, the OPU ISA is designed to be polymorphic. This implies that a same compiled binary can run on any different OPU cores, even if they are designed to operate on different data types (e.g. 4-bit or 8-bit integers).

Buffers

Similarly to how typical ISAs define registers, the OPU ISA defines four different buffers named *fm*, *ker*, *bias*, and *ofm*. In contract to general purpose registers, however, each of these buffers is intended to store only one type of operand. This specialization greatly simplifies the implementation of efficient OPU cores. In particular, when executing a neural network layer, *fm* contains the input features of the layer, *ker* contains its input weights, *bias* stores the input bias values. The *ofm* buffer is employed to store the partial sums of the layer. As detailed in Table 2.1, the exact characteristics of these buffers are for the most part implementation-dependent.

Buffer	Dimensions	Data format
fm	2048×64	Implementation-defined
ker	$32 \times$ `NUM_MAC`	Implementation-defined
bias	64×1	Implementation-defined
ofm	2048×64	Implementation-defined

Table 2.1: Characteristics of the OPU buffers.

Operations

The OPU ISA supports three different types of operations: *load*, *compute*, and *store*. As shown in Figure 2.1, these operations read values from memory/buffers, perform some useful function, then write the result to another memory location. In contrast to general-purpose ISAs, these operations generally take hundreds of clock cycles to complete. Their exact effect

on the buffers and memory is specified by numerous parameters, which are stored in special parameter registers. These parameter registers can be set at runtime, between the execution of different neural network models, or at any time during the execution of a single model.

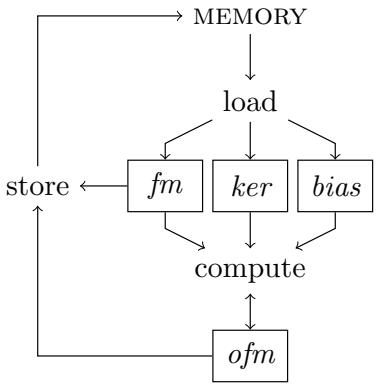

Figure 2.1: Programmer's model of the OPU ISA.

Load operation

A *load* operation performs a data transfer from memory to *fm*, *ker*, or *bias*. The *ofm* buffer cannot be directly written. Register parameters defines how this transfer should be done. Depending on their values, a single *load* operation can write data into one or more of these three buffers. Naturally, various access pattern are available. Most notably, an entire slice of a 3D array can be loaded from memory at once as a single operation. In general, a single *load* operation can transfer a total of multiple megabits of data from memory to the on-chip buffers.

Compute operation

The *compute* operation mainly performs dot products between the rows of *fm* and *ker*. Depending on the values of the parameter register, this operation can add the bias values to the result of this dot product, or instead add the values of the *ofm* buffers for accumulation. The *compute* operation can also scale the operands by any power of 2, which is often required for quantized models.

A single *compute* operation can perform a multitude of dot products, which generally correspond to millions of individual multiply-accumulate operations. In addition, various data broadcast mechanisms are available to facilitate data reuse, and thereby greatly improve efficiency.

Store operation

The *store* operation transfers data from *ofm* to the memory. The content of other buffers cannot be directly written to memory. In addition to simple data transfers, all functions other than convolutions and dot products are also performed by the *store* operation. Notably, this operation supports data padding, pooling, activation, and residual addition. The order in which these post-processing steps are preformed can be specified using various register parameters.

Instruction flow

The current version of the OPU ISA employs a dataflow architecture. In contrast to more common control flow schemes where instructions are stored and executed sequentially, an OPU program is equivalent to a directed acyclic computational graph. The nodes of the graph correspond to operations and the edges represent data dependencies. Naturally, an instruction can only start executing if all its parents have been fully processed. For illustration, Figure 2.2 shows the computational graph of a simple program. In that example, the *compute* operations are only allowed to start after their corresponding *load* operation is complete. Likewise, for correct accumulation of results, the second *compute* must be executed after the first one. The final *store* operation can only be performed when the results have been produced by the two *compute* operations.

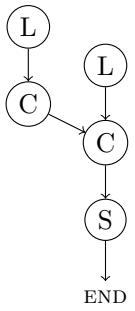

Figure 2.2: Example of dataflow graph (L: load, C: compute, S: store).

In OPU, data dependencies are all statically detected by the compiler and embedded into

the executable binary. This strategy was initially chosen in order to simplify the implementation of instruction-level parallelism into OPU cores [39]. However, future versions of the OPU ISA will likely implement a more traditional control flow scheme. This will allow for a more efficient instruction encoding, as well as a notable simplification of the supporting software toolchain.

Instruction blocks & sub-instructions

As described previously, all three operations of the OPU ISA are largely parametrized. Encoding all of these parameters in each instruction would thus result in impractically large executable binaries. To solve this issue, OPU programs are divided into instruction blocks, each containing one or more sub-instructions. In the executable, each instruction block corresponds to an operation (i.e. load, compute, or store) that needs to be performed. An instruction block always starts with a so-called C-type instruction that specifies the type of the operation and its data dependencies, as required by the dataflow architecture. The rest of the instruction block consists of U-type instructions which set the parameter values for this operation. Parameters that are not set in the current instruction block maintain their last assigned value. Because constant parameters do not need to be specified repeatedly, and due to the intrinsic regularity of typical neural network layers, executable length can generally be reduced tenfold.

2.3 Compiler toolchain

As described in [39], a compiler has been developed for the OPU ISA. This compiler converts a high-level representation of a neural network into an OPU executable binary. Currently, it accepts models both in .pb format as generated by the popular TensorFlow framework [30], and in the Open Neural Network Exchange format [4].

In order to ease as much as possible to process of deploying machine learning models, the compiler automatically transforms the provided model's data type into the proper format for the target OPU implementation. Generally, this requires the quantization of IEEE 754 [19] floating-point numbers to a smaller fixed- or floating-point representation.

As mentioned previously, data dependency identification and instruction scheduling are also performed by the compiler. In addition, for maximum performance, the compiler can apply various custom optimizations. These include operation fusion, reshaping of layers, and layer grouping. These optimizations allow to maximize the utilization of the resources of the processor, thereby increasing its effective throughput.

2.4 Existing implementations

Excluding the implementation described in the present document, two fully-operational cores implementing the OPU ISA have been designed so far. Both cores are described in [39].

The first core (referred to as OPU1024 in the rest of this document) is implemented with `NUM_MAC` = 1024 on a Xilinx XC7K325T, whereas the second core (referred to as OPU4096 henceforth) is implemented on a significantly larger Xilinx XC7Z100 with `NUM_MAC` = 4096. These two cores are designed to operate on 8-bit fixed-point numbers. Because they do not use traditional IEEE 754 floating-point numbers, models need to be quantized by the compiler, and possibly fine-tuned, before being mapped to the hardware. The data formats of the employed buffers are listed in Table 2.2.

Buffer	Data format
fm	8-bit fixed-point
ker	8-bit fixed-point
bias	16-bit fixed-point
ofm	16-bit fixed-point

Table 2.2: Data formats of prior implementations.

In order to maximize performance, these implementations both have two copies of the *fm*, *ker* and *bias* buffers. This allow the execution in parallel of one load operation and one compute operation. This parallelism effectively hides the latency associated with the transfer of data from memory to the core.

Both implemented cores can operate at a maximum clock frequency of 200MHz. For the

inference process of AlexNet [25], the smaller core provides a throughput of 354GOPs with an average power consumption of 16.5W[1]. The larger core reaches 1218GOPs while only consuming 17.7W[1]. This corresponds respectively to a 6× and a 19× better power efficiency compared with a Nvidia Titan Xp GPU. In addition, in spite of their high flexibility, these OPU cores can outperform automatically-compiled network-specific FPGA accelerators in the literature. These initial promising results highlight the relevance of the OPU ISA for the acceleration of deep neural network inference.

[1]Power consumption including development board and power supply.

Chapter 3

Field-Programmable Gate Arrays

The OPU ISA is currently mainly intended to be used in overlay processors implemented on field-programmable gate arrays (FPGAs). As their name indicates, FPGAs are integrated circuits that can be reconfigured "in the field", after being manufactured. This property makes these devices especially suited for prototyping, as well as for real-life applications due to their drastically reduced cost compared to full custom ICs. Though their speed is not as high as the latter, FPGAs can provide much better performance than regular microprocessors for a large variety of tasks. For this reason, FPGAs have had tremendous success for audio/video processing, telecommunication, as well as for medical, automotive, and aerospace applications.

Due to their history, the layout of modern FPGAs is mainly optimized for traditional digital signal processing tasks. However, over the last decade, the use of FPGAs for more generic

application has gained much attention by academia and the industry. Notably, FPGAs now represent a common platform for machine learning acceleration. Likewise, the use of FPGAs to replace CPUs in the datacenters for compute-hungry tasks has become increasingly popular.

This chapter begins with a basic overview of the inner workings of FPGAs. Then, more specific details of these devices are explored, while highlighting the major differences that exist between the FPGAs designed and manufactured by two major players of the programmable logic market: Xilinx Inc., used for the existing OPU cores, and Microsemi Corporation, employed here for the new implementation.

3.1 Overview

The field-programmability of FPGAs is the result of two main components of these devices: programmable logic cells, and programmable routing resources.

The logic cells are circuits designed to be able to emulate any logic function. To achieve this, these cells generally implement logic using small memories called lookup tables (LUTs) that store the truth table of the required function. An n-input LUT thus contains 2^n stored bit, and can implement any n-ary function. Because the content of these memories can be modified, it is possible to change the behavior of the logic cells as needed. Given this,

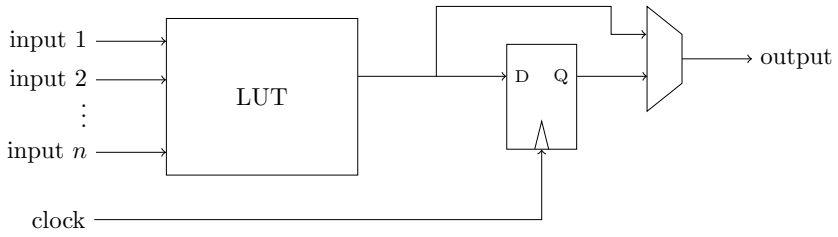

Figure 3.1: Typical logic cell organization.

the difference in speed between FPGAs and ASICs is explained by the fact reading these memories is much slower than an equivalent CMOS gate.

Lookup tables can only be used to implement combinational logic. For generating sequential logic, the output of each LUT is generally connected to a traditional flip-flop or latch. This sequential element can optionally be bypassed as needed by the user. For illustration, Figure 3.1 gives a high-level view of typical logic cell.

For interconnection between these logic cells, FPGAs provide reconfigurable routing resources. These generally consist of many layers of wires arranged in a regular pattern, with programmable switch boxes at appropriate locations. Figure 3.2 illustrates an example of how logic cells could be interconnected using configurable routing.

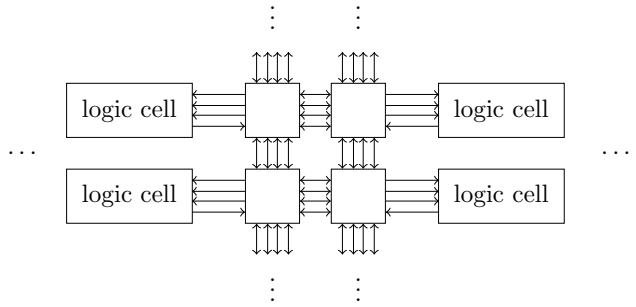

Figure 3.2: Abstract schematic of an FPGA device.

3.2 Lookup table size

Different FPGA devices are implemented using different sizes of lookup tables. This design decision can impact significantly the resulting performance of the mapped circuits.

As mentioned previously, an n-input LUT can simulate any combinational n-ary function. Functions with more than n inputs, however, always require multiple stages of LUTs in order to be implemented. Because of this, larger LUTs (i.e. LUTs with more inputs) tend to reduce the numbers of logic stages between registers compared to smaller LUTs. Larger LUTs have thus the potential to allow for faster mapped designs.

Nevertheless, larger LUTs also tend to be less area-efficient. Indeed, larger LUTs offer a coarser granularity for mapping circuits to logic cells, which generally leads a less efficient utilization of the LUTs. This wastes valuable silicon area and can thus limit the capabilities

of the FPGA device. In addition, poor utilization of the LUTs causes logic cells to be unnecessarily farther apart from each other. As a result, wires may be in average longer than actually required, thereby reducing the maximum clock frequency of the design.

Because of these considerations, some digital circuits map better to small LUTs, while others map more efficiency to larger LUTs. Equivalently, this fact implies that the optimal gate-level implementation of a given functionality depends on the size of the LUTs it will be mapped to. Notably, one should expect the optimal number of pipeline stages to be smaller when using large LUTs than with smaller ones.

Microsemi FPGAs are implemented using traditional 4-input LUTs exclusively. In contrast, Xilinx uses pairs of 5-input LUTs, arranged as shown in Figure 3.3. Because both individual LUTs of each pair share the same inputs, these cannot be used fully independently from each other as they can only implement two 5-ary functions of the same arguments. Thanks to the extra multiplexer, however, one pair of such LUTs can alternatively simulate a single 6-ary function. In addition, Xilinx FPGAs have extra logic around their LUTs so that two adjacent pairs of 5-inputs LUTs can be used as one 7-input LUT, and four pairs can act as a large 8-input LUT.

Given the difference in the size of the LUTs of Microsemi FPGAs and Xilinx FPGAs, the optimal way to program these devices will naturally be different. This motivates the need

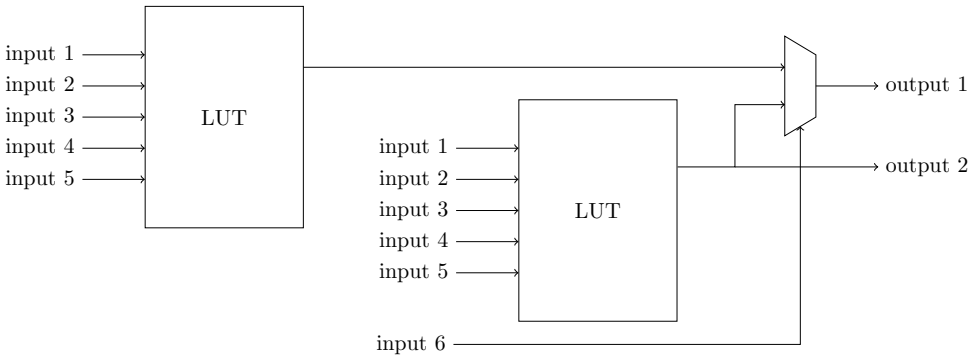

Figure 3.3: Xilinx's LUT arrangement in 7 Series FPGAs.

for implementing a new OPU core tailored for Microsemi FPGAs.

3.3 Technology

As explained, a lookup table is a small memory that stores a truth table, and therefore effectively acts as a logic gate. These memories can themselves be implemented physically using different kinds of memory technology, a design decision that greatly impacts the characteristics of the resulting FPGA device. In particular, Xilinx uses SRAM technology to implement its LUTS, while Microsemi employs flash memory. Other technologies such as anti-fused FPGA also exist.

SRAM-based lookup tables are implemented using an array of SRAM cells. In general, each such cell is made of 6 transistors arranged as shown in Figure 3.4. Because it stores information on a pair of cross-coupled CMOS inverters, an SRAM cell needs to be powered

at all time in order to keep its stored bit. From a system-level view, this property implies that the FPGA device must be reprogrammed (i.e. the appropriate data must be written into the LUTs) every time it is powered on. SRAM-based FPGAs must therefore also always be accompanied by an external nonvolatile storage device for storing the configuration while the FPGA device is not powered.

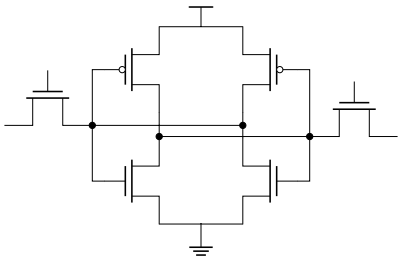

Figure 3.4: Typical SRAM cell.

Flash-based FPGAs employ nonvolatile storage, such as the NAND cell shown in Figure 3.5, for the implementation of their lookup tables. Naturally, with this approach, configuration can be kept on the device at all time, and reconfiguration is not required at startup. Compared to their SRAM-based counterparts, flash-based FPGA present a few advantages in terms of security, reliability, and power consumption.

Security With an SRAM-based FPGA, the configuration bitstream needs to be copied from an external nonvolatile memory to the FPGA for programming. Because of this, an attacker could very easily capture this bitstream as it is being transferred and, if it is not

properly encrypted, steal valuable intellectual property. By contrast, with an flash-based FPGA, configuration information never leaves the device after an initial programming. In addition, it is generally considered more difficult to read the state of a NAND flash cell than an SRAM cell using inspection techniques. For these two reasons, flash-based FPGAs are preferred when security of the mapped design matters.

Reliability Nonvolatile flash technology is significantly more resilient to radiations than SRAM. In other words, the probability that a configuration bit randomly flips is much lower on flash-based FPGA than SRAM-based ones. This makes the former more adequate for applications where large amount of radiation are present, such as in airplanes and spacecraft, or for highly critical applications on the ground. Furthermore, because they do not require to be reconfigured at each startup, flash-based FPGAs can resume normal operation much more quickly than SRAM-based devices after an unexpected power failure. As many promising deep learning applications can be considered mission- or life-critical, notably in the medical and automotive sectors, this enhanced reliability is valuable for OPU cores.

Power consumption In addition to the obvious power saving at startup, flash-based FPGAs consume less power during operation than an equivalent SRAM-based FPGA. Indeed, the leakage current of a SRAM cell can be quite significant. The static current of flash-based cells is negligible by comparison. Based on this fact, Microsemi estimates that their flash-based FPGAs consume 30% to 50% lower total power than competitive FPGAs [8].

Such power reduction, if reached in practice, could be extremely advantageous for OPU or any other deep learning accelerator.

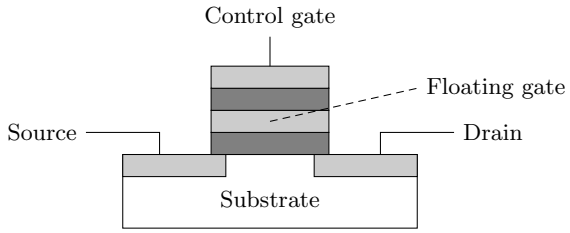

Figure 3.5: Cross section of a typical NAND flash cell.

3.4 Hard resources

Because some functionalities are commonly needed in FPGA-mapped design, virtually all vendors embed into their FPGA fabric various *hard* blocks, i.e. circuits efficiently implemented using CMOS gates and whose behavior therefore cannot be modified after manufacturing. In the context of this thesis, the two most important of these blocks are memories and arithmetic units.

Memory

Most FPGA-mapped designs require the ability to store decent amount of data on the device. Implementing such feature using the sequential elements of the logic cells would result in extremely inefficient design. For this reason, all FPGAs provide memory resources in

their fabric. These are generally composed of low-capacity SRAM arrays scattered across the device. If required, these small SRAM blocks can be interconnected using regular fabric resources in order to create wider or taller memory blocks.

As described in [37], Xilinx's devices contain two types of memory: block RAM (BRAM) and Distributed RAM. Since the lookup tables are themselves small SRAM arrays, it is relatively inexpensive for Xilinx to allow the user to employs some of these lookup tables as random access memory. Naturally, a same LUT cannot be used at the same time for implementing logic functions and for writable RAM. In Xilinx FPGAs, roughly one third of the LUTs are extended to be used as RAM, thereby forming so-called Distributed RAM resources. The size of the smallest RAM block that can be created in this way is 32×16 bits. Larger memories require multiple such blocks. However, combining a large number of these blocks is generally suboptimal due to the interconnections needed to combine these blocks.

For larger memories, Xilinx provides BRAMs, each of which can store as much as 36Kb. In contrast to Distributed RAM, BRAMs can be dual-ported, meaning that they can perform two read and two write operations at each cycle. In addition, the shape of each BRAM can be configured for more flexibility.

As Microsemi's lookup tables are not implemented with SRAM cells but with NAND flash cells, they cannot be used as read/write memory. Nevertheless, Microsemi FPGAs

provides two types of SRAM memory blocks, implemented besides the LUTs. As described in [31], these two types are μSRAM for small memories, and LSRAM for larger ones. Each μSRAM has a size of 64×12 bits, whereas each LSRAM can store a total of 20Kb. Just like Xilinx's BRAMs, LRAMs are dual-ported and can be configured into different shapes as needed by the user.

Arithmetic

Because FPGAs have mostly been targeted to signal processing tasks, a major components of their implementation are the so-called DSP blocks. These hard resources are designed to efficiently perform wide integer multiply-accumulate operations. Such operation is indeed ubiquitous in digital filters (hence the name of the DSP blocks). These are also greatly important for the implementation of linear algebra operation of machine learning models.

Current Xilinx FPGAs use *DSP48E1 Slices* for implementation of their DSP blocks. The exact functionalities of these hard resources are fully described in the manual [38]. Figure 3.6, copied from this manual, presents the basic architecture of a Xilinx DSP block. The main components of this block are:

i. One 25-bit adder/subtractor.

ii. One 25×18 two's complement multiplier.

iii. One 3-operand 48-bit adder/subtractor.

iv. One 48-bit pattern detector.

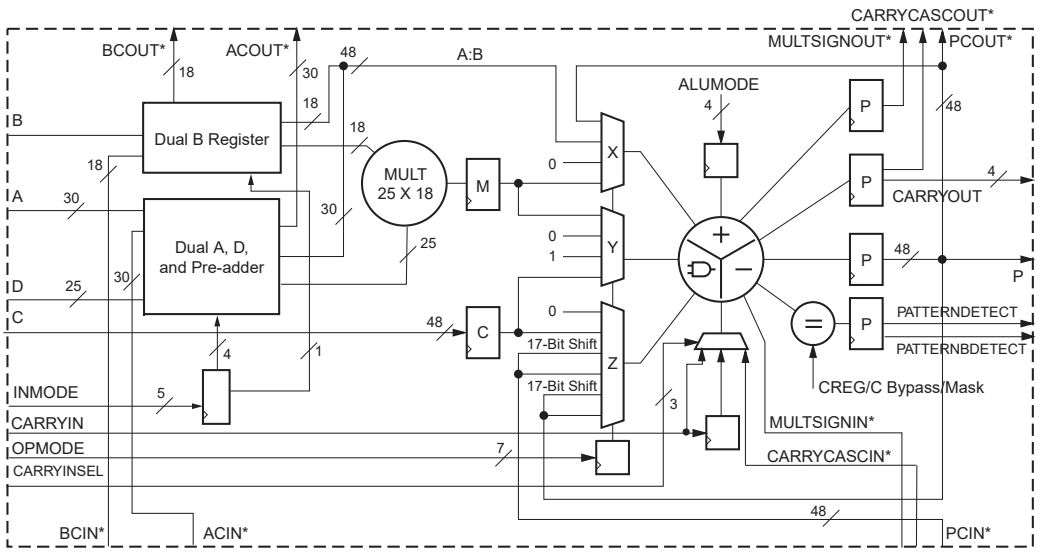

Figure 3.6: Basic Xilinx DSP48E1 slice functionality [38].

Various other mechanisms are implemented to provide more flexibility. For instance, the 48-bit adder can be optionally configured to perform any bitwise operation with its two operands. The carry chain of this adder can be split to decompose it into two smaller 24-bit adders. For maximum performance, most of the elements can be bypassed, and multiple pipeline stages can be optionally activated inside of the DSP.

On the FPGA, the DSPs are arranged in rows, with dedicated connections between adjacent DSP blocks. The signals using these routing resources are marked with * in Figure 3.6. Thanks to these, complex functions can be implemented efficiently using chains of DSPs with no usage of the general routing resources of the FPGA.

The DSP block of Microsemi FPGAs, marketed as *Math block*, are similar to the one designed by Xilinx. Details of Microsemi's DSPs are given in the manual [31]. Figure 3.7, extracted from this manual, shows a simplified schematic of the architecture of these DSPs. The main elements of this block are:

i. One 18-bit adder/subtractor.

ii. One 18×19 two's complement multiplier.

iii. One 3-operand 48 adder/subtractor.

The most important difference compared to Xilinx DSPs is the smaller size of their multiplier. The Microsemi DSP also lack a pattern detector and a the ability to apply arbitrary bitwise functions. One should however note that these two functionalities are typically easy to implement using only LUTs. Another major difference between the two vendors is that a single Microsemi DSP can perform

$$x_0 \times y_0 + x_1 \times y_1$$

where x_0, x_1, y_0, y_1 are all 9-bit integers. In contrast, a single Xilinx DSP can only produce the two products

$$x_0 \times y_0 \quad \text{and} \quad x_0 \times y_1.$$

As such, Mircosemi's DSPs are more adequate for computing dot products.

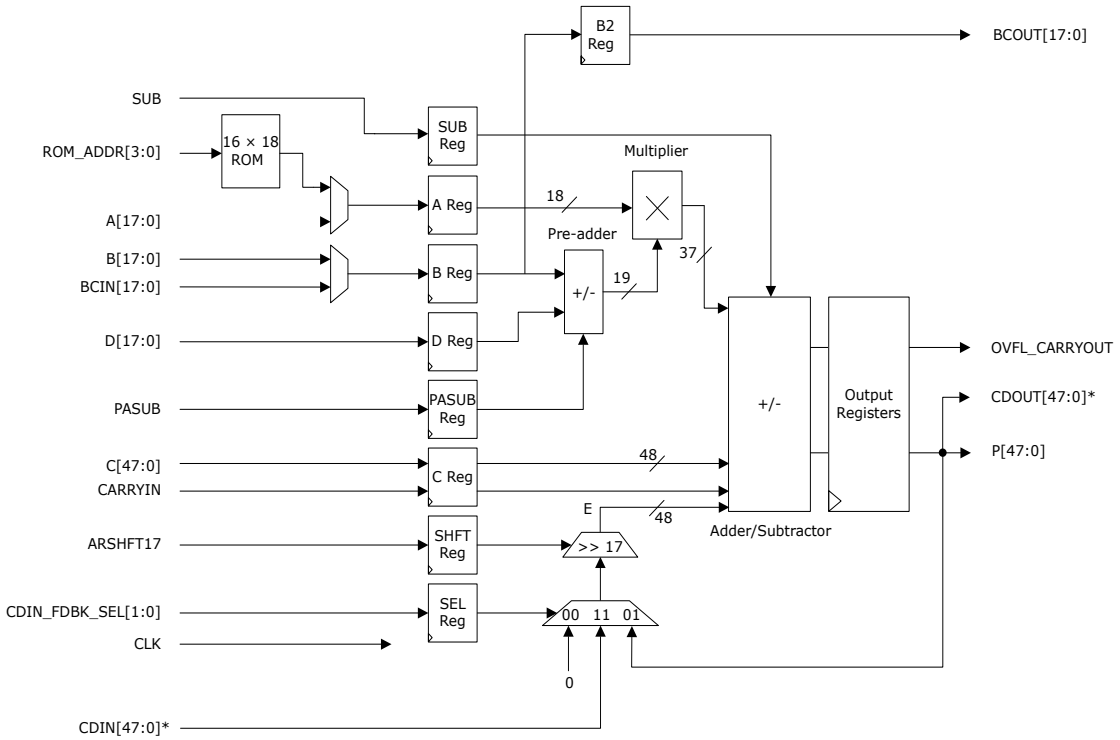

Figure 3.7: Basic Microsemi Math block functionality [31].

Just like Xilinx DSPs, pipeline stages can be activated inside Microsemi's DSP to increase the maximum frequency. Microsemi's DSPs are also arranged in rows with dedicated routing resources between adjacent blocks for maximum performance.

Chapter 4

Implementation details

Given the potential advantages of flash-based FPGA listed in Chapter 3, an implementation of OPU on such a FPGA could be valuable. In this work, an OPU core is thus implemented on a Microsemi PolarFire 300 (MPF300). This specific device contains 300 000 4-input LUTs and 954 DSPs. This design is the third complete OPU core, and the first implementation designed for on an FPGA that is not manufactured by Xilinx. In this way, this work supports the goals of OPU regarding standardization and platform intercompatibility.

The new core is derived from the existing OPU1024 [39], and, as such, its general architecture is similar. Nevertheless, multiple changes have been made to this original design in order to fit the MPF300 and to provide more flexibility. Notably, because the MPF300 utilizes smaller LUTs, several critical modules have been manually optimized to provide optimal mapping onto the FPGA. These hand-optimization reduced by approximately 60 000

the total number of LUTs needed in the core compared to when using the same RTL implementation as OPU1024. As detailed in the rest of this chapter, in addition to these optimizations, the dot product unit of the core has been completely redesigned.

4.1 Data type

As mentioned in Chapter 2, OPU is a polymorphic ISA that can describe operations on different types of data formats. Even though different OPU cores can be implemented for any data format, they all can execute the same compiled binary. The two existing OPU cores both have been designed to operate on 8-bit fixed-point numbers. This data format is used because it is significantly easier to perform arithmetic on narrow integers than on the IEEE 754 floating-point numbers that are used for training. Such a quantization is acceptable because it generally only negligibly impacts the accuracy of the machine learning models [21, 14]. For these reasons, 8-bit fixed-point representation is one of the most popular data format used in modern accelerators [23, 6, 36, 1].

The major drawback of fixed-point representations is that models need to be fine-tuned after quantization in order to maintain a high accuracy [35, 27], which can be highly compute-hungry. In contrast, narrow floating-point representations can generally yield high accuracy without any fine-tuning step [28, 24]. In this context, many different such representations have been proposed. Some, such as [35, 26], simply extend the IEEE 754 standard to a lower

bit count, whereas others, such as [22, 11], take different approaches than current standards. These methods can all provide without any fine-tuning a similar accuracy than a fixed-point format of the same width. In addition, narrow floating-point and fixed-point arithmetic circuits tend to require a similar silicon area. As such, floating-point representations appear to be overall slightly superior.

For this reason, the new OPU core proposed in this work uses a floating-point representation of numbers. This is thus the first OPU core capable of operating on non fixed-point data types. The employed format is here a straightforward extension of the IEEE 754 standard, identical to the one described in [35]. For given positive x and y widths, a number z is represented by a sign bit z_s, a y-bit exponent z_e, and an x-bit positive mantissa z_m arranged as follows:

$$\underbrace{0}_{z_s}\underbrace{00\ldots00}_{z_e}\underbrace{00\ldots00}_{z_m} \tag{4.1}$$

This representation, stored on $x + y + 1$ bits, is referred to as MxEy. The value of any represented number z is then given by

$$z = (-1)^{z_s} \times 2^{z_{e^\star}} \times z_{m^\star} \tag{4.2}$$

where

$$z_{e^\star} = \begin{cases} 0 & z_e = 0 \\ z_e - 1 & \text{otherwise} \end{cases}, \quad \text{and} \quad z_{m^\star} = \begin{cases} z_m/2^x & z_e = 0 \\ z_m/2^x + 1 & \text{otherwise} \end{cases}. \tag{4.3}$$

In this work, to ease the interactions with the memory system, only 8-bit representations are considered. From the study in [35], two different 8-bit MxEy-like formats can provide enough accuracy for typical neural networks without fine-tuning: M4E3 and M5E2. Among these two formats, M5E2 tends to yield a slightly better model accuracy than M4E3. Nevertheless, the accuracy provided by both formats is acceptable. Both M4E3 and M5E2 are thus considered here for the implemented core.

4.2 Dot product unit

The most compute-hungry operations of deep neural networks are linear algebra operations that can be expressed as sequences of dot products: convolutions and matrix-vector multiplications. As such, the dot product unit of an OPU core is its most significant module. As this unit alone can determine the performance of the OPU core, its optimization is of great importance.

In OPU, the dot product unit can generally be implemented as an array of 16 *processing elements* (PEs) that each computes, every cycle, four n-dimensional dot products. Figure 4.1 shows the generic structure of a single PE. Notice that, even though the PE performs four dot products, it has only four different vector operands: $\mathbf{a}, \mathbf{b}, \mathbf{c}, \mathbf{d}$. Each of these operands

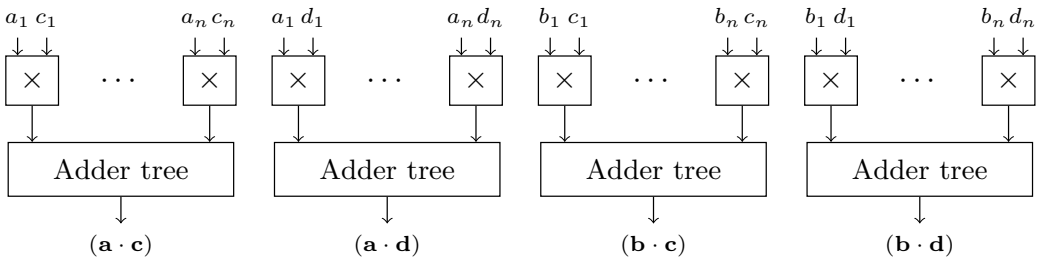

Figure 4.1: Generic structure of a single processing element.

is used for two dot products. According to the ISA specifications, the dimension n of these vectors depends on the `NUM_MAC` parameter of the considered core and is given by

$$n = \frac{\texttt{NUM\_MAC}}{64}.$$

In that configuration, each PE performs $4n$ multiply-accumulate operations per cycle. In total, the OPU core can thus compute $16 \times 4n = \texttt{NUM\_MAC}$ multiply-accumulate operations per cycle. For the OPU core described in this work, `NUM_MAC` is set to 2048. Each individual dot product is thus performed on 32-element vectors.

For this core, the vector elements are 8-bit floating-point numbers, hence $4 \times 32 = 128$ floating-point multipliers are needed in each PE. The generic structure of such a multiplier is shown in Figure 4.2. In order to greatly reduce the complexity of the dot product unit, the multipliers are designed to produce their results in fixed-point representation. The results are accumulated using trees of fixed-point adder, without performing any intermediary rounding operation. As a consequence, the accuracy of the computed dot products is improved, which

helps limit the negative impact of quantization.

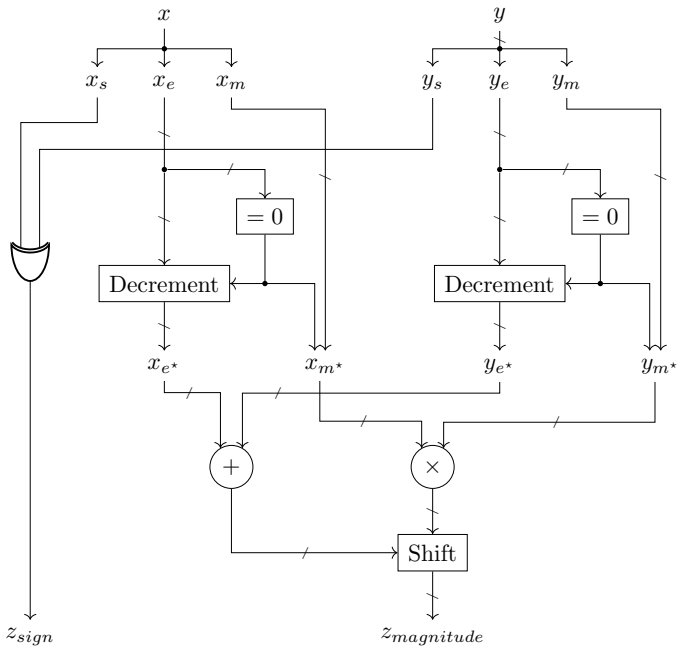

Figure 4.2: Floating-point multiplier with fixed-point output.

The final four outputs of each PE are kept in two's complement fixed-point format for further operations, and are only converted back to 8-bit floating-point when written to external memory. The formats of the various buffers of this OPU core are thus as listed in Table 4.1.

Buffer	Data format
fm	8-bit floating-point
ker	8-bit floating-point
bias	16-bit fixed-point
ofm	16-bit fixed-point

Table 4.1: Data formats of proposed implementation.

Mantissa multiplication on DSPs

Given the different components required to implement a floating-point multiplier, as shown in Figure 4.2, it is reasonable to consider using the DSPs of the FPGA for the multiplication of the mantissas. Indeed, integer multipliers can be quite costly when implemented using LUTs, whereas DSPs contain very efficient multipliers. As the multiplier of the DSPs is much wider than the considered mantissa, one DSP could be used for more than one product. In particular, in this section, the utilization of a single DSP for two and for four products are both studied.

Two multiplications per DSP

When configured in SIMD mode [31], a single DSP of the MPF300 FPGA can perform two 9×9 signed multiplications. In M5E2 format, after adding the implicit unit bit, as described in (4.3), the width of the mantissa is 6 bits. It is thus possible to perform two mantissa multiplications in one DSP. Naturally, this is equally feasible using the M5E3 format, but this option is not studied here as M5E2 has been shown to yield better accuracy.

When using the DSPs in that way, the rest of the design must use LUTs, including the adder tree, the shifters, and exponent adder. The exact resource usage of this approach is detailed in Table 4.2. One can notice that implementing 16 PEs using this strategy would require as much as 1024 DSPs, which is more than the 954 available DSPs. Therefore, some of the multipliers need to implemented using LUTs, each of them requiring 83 LUTs. In total, the 16 PEs of the implementation would thus require

$$16 \times 10404 + (2048 - 954 \times 2) \times 83 = 178084 \text{ LUTs.}$$

This would represent 60% of the 300K available LUTs, which does not leave enough resources to implement the rest of the OPU core. Unless the targeted number of multiply-accumulate operation per cycle `NUM_MAC` is reduced, this strategy is thus not resource-efficient enough to be applied here.

Component	Count	LUTs	DSPs
Exponent add.	128×	3	-
Mantissa mult.	128×	0	0.5
Shifter	128×	41	-
Adder tree	4×	1065	-
Other	128×	4	-
Total		10404	64

Table 4.2: Resources per PE with two mantissa multiplications per DSP.

Four multiplications per DSP

In [35], a strategy for implementing four 5-bit multiplications in a singe Xilinx DSP is proposed. This paper presents very promising result, including the fact that this approach can yield 1.5× larger computation throughput compared to other FPGA accelerators. Given these promises, the proposed approach is here tested on the MPF300 device, whose DSPs contain slightly narrower multipliers. Indeed, whereas a Xilinx DSP can compute a 25×18 signed multiplication, a single DSP of the MFP300 device performs

$$P = (B + D) \times A + C \tag{4.4}$$

where the result P is a 48-bit two's complement number, and the operands A, B, C, D are respectively 18-, 18-, 48-, and 18-bit two's complement numbers. The multiplier of the Microsemi DSP is thus of size 18×19. Note that for this experiment, the M4E3 format is preferred as its mantissa are one bit narrower than the M5E2 format, which makes it easier to perform four products per DSP.

In general, a single wide unsigned multiplier can be used to perform more than one narrow multiplication thanks to the distributive property of multiplication. Given four different operands (a, b, c, d), four multiplications can be performed by multiplying the values $(a2^n + b)$ and $(c2^n + d)$. Indeed, it can be noticed that the result

$$(a2^n + b) \times (c2^m + d) = (a \times c)2^{n+m} + (b \times c)2^m + (a \times d)2^n + (b \times d) \tag{4.5}$$

contains the products $a \times c, a \times d, b \times c$, and $b \times d$. These products, however, can only be unambiguously extracted from the result if they do not overlap.

One can thus easily derive that an unsigned multiplier of size $W_1 \times W_2$ can be used to compute the products $a \times c, a \times d, b \times c, b \times d$, where a, b, c, d are unsigned integer of widths W_a, W_b, W_c, W_d, if and only if

$$\exists m, n \in \mathbb{N} : \begin{cases} n \geqslant W_b \\ n + W_a \leqslant W_1 \\ m \geqslant W_d \\ W_c + m \leqslant W_2 \\ W_d + W_b \leqslant n \\ n + W_d + W_a \leqslant m \\ W_b + W_c + m \leqslant n + m \\ n + m + W_a + W_c \leqslant W_1 + W_2 \end{cases} . \tag{4.6}$$

In the MPF300 device, because the 18×19 multiplier is designed to operate on two's complement numbers, the usable operand widths are here $W_1 = 17$ and $W_2 = 18$. After solving (4.6) for this particular size of multiplier, the found possible solutions for (W_a, W_b, W_c, W_d) are

$$(3,3,3,3)\quad (4,3,3,3)\quad (5,3,3,3)\quad (6,3,3,3)\quad (3,3,4,3)\quad (3,4,3,3)$$
$$(3,3,3,4)\quad (3,3,4,4)\quad (4,3,4,3)\quad (4,4,3,3)\quad (4,3,3,4)\quad (5,4,3,3)$$
$$(3,4,4,3)\quad (3,5,3,3)\quad (3,4,3,4)\quad (4,5,3,3)\quad (3,6,3,3)^{1}.$$

The absence of $(5,5,5,5)$ in this solution set implies that, contrary to what is possible in a Xilinx DSP, it is impossible to perform four 5-bit multiplications using exclusively one Microsemi DSP. Therefore, one cannot compute four M4E3 mantissa products in one DSP. The best possible solution is here $(4,5,3,3)$, which correspond to performing two 4×3 and two 5×3 multiplications using only one DSP. As this is not sufficient for computing the products of the mantissas, a few small extra multipliers must be implemented using LUTs.

The C operand of the DSP can be employed here to save an addition that would have otherwise required LUTs. In addition, the pre-addition of the operand B and D in (4.4) is used in order to fully exploit the width of the available multiplier. All in all, the four required products ac, ad, bc, bd are obtained as follows:

[1]Solutions with operands narrower than 3 bits are omitted here.

i. The four following extra terms are evaluated:

$$\alpha = ac_{4:3} + 2a_4c_{2:0},$$

$$\beta = bc_{4:3},$$

$$\gamma = ad_{4:3} + 2a_4d_{2:0},$$

$$\delta = bd_{4:3}.$$

ii. The operands of the DSP are formed:

$$A = a_{3:0}2^8 + b,$$

$$B = c_{2:0}2^{14} + d_{2:0},$$

$$C = \alpha 2^{26},$$

$$D = c_{2:0}2^{14}.$$

iii. One DSP is used to compute

$$P = P_{32:0} = A \times (B + D) + C$$

iv. The four multiplications are obtained with

$$ac = P_{32:23},$$

$$bc = \beta 2^3 + P_{22:15},$$

$$ad = \gamma 2^3 + P_{14:8},$$

$$bd = \delta 2^3 + P_{7:0}.$$

This approach allows to compute four mantissa products with one DSP and a few LUTs. The total resource utilization per PE of this approach is shown in Table 4.3. As can be seen, the 954 DSPs of the MPF300 are enough to implement the targeted 16 PEs. However the number of LUTs that would be needed is equal to $16 \times 14728 = 235648$ LUTs, which represents 78% of the available resources. As with the previous approach, this strategy does not leave enough LUTs to implement the rest of the design, and is therefore unusable in practice. Remarkably, performing two mantissa multiplications per DSP is more resource-efficient than attempting to perform four multiplications per DSP. The result of [35] are thus not applicable for Microsemi FPGAs.

Component	Count	LUTs	DSPs
Exponent add.	128×	4	-
Mantissa mult.	128×	14	0.25
Shifter	128×	54	-
Adder tree	4×	1250	-
Other	128×	4	-
Total		14728	32

Table 4.3: Resources per PE with four mantissa multiplications per DSP.

Full-precision multiplication

It is clear that using the DSPs exclusively for mantissa multiplication requires a large number of extra lookup tables per PE. In particular, the shifters needed to align the floating-point products before their addition are costly. Indeed, they account for as much as 47% of the

needed LUTs. The adder trees, too, contributes significantly to the large resource utilization, with 33% of the LUTs utilized for this purpose. A PE design that removes the need for the product alignments and addition using LUT is thus studied here. Essentially, this approach consists in naively performing the floating-point multiplications using full-precision fixed-point representations of the operands.

For this approach the DSPs are employed in so-called DOTP mode. As explained in the FPGA's manual [31], in this configuration, each DSP can compute

$$P = \begin{cases} (B_1 + D_1) \times A_1 + (B_2 + D_2) \times A_2 + C + E + F & \text{if } \overline{S_1} \wedge \overline{S_2} \\ (B_1 + D_1) \times A_1 - (B_2 + D_2) \times A_2 + C + E + F & \text{if } \overline{S_1} \wedge S_2 \\ (B_1 - D_1) \times A_1 + (B_2 - D_2) \times A_2 + C + E + F & \text{if } S_1 \wedge \overline{S_2} \\ (B_1 - D_1) \times A_1 - (B_2 - D_2) \times A_2 + C + E + F & \text{if } S_1 \wedge S_2 \end{cases} \tag{4.7}$$

where the data formats of the operands are given in Table 4.4. The operand E is hard-wired to the output P of the adjacent DSP. This operand thus cannot accept arbitrary input from LUTs in the fabric.

It is worth noticing that the magnitude of a floating-point number represented in M5E2 format can be represented in full-precision with only 8 bits. As the inputs $A_{1,2}$ and $B_{1,2}$ are wider than this, it is therefore possible to use a single DSP to compute the sum of the products of two pairs of M5E2 numbers. Naturally, this requires that these four M5E2 operands be converted to full-precision sign-and-magnitude beforehand, using a few LUTs.

Operand	Width	Format
A_1, A_2	9 bits	two's complement
B_1, B_2	9 bits	two's complement
C	39 bits	two's complement
D_1, D_2	9 bits	two's complement
E	39 bits	two's complement
F	1 bit	unsigned
S_1, S_2	1 bit	unsigned

Table 4.4: Operands of a DSP in DOTP mode.

Note however that it is not necessary to convert the four numbers to two's complement representation as the inputs S_1 and S_2 and the pre-adders can be exploited to deal correctly with sign-and-magnitude operands. This is done by setting B_1 and D_2 to zero. Indeed, the operation (4.7) of the DSP then becomes

$$P = (-1)^{S_1} \times D_1 \times A_1 + (-1)^{S_2} \times B_2 \times A_2 + C + E + F. \tag{4.8}$$

Accordingly, the inputs S_1 and S_2 are set to the sign of the products (i.e. XORing of the operands' signs) and the 8 low significant bits of the inputs A_1, D_1, A_2, B_2 are used for the magnitude of the operands. Figure 4.3 summarizes how a M5E2 two-element dot product can be computed using one DSP and a few LUTs using this method.

Implementing 16 PEs using this method would require more DSPs that what is available on the considered FPGA device. Therefore, some of the multiply-accumulate operations must be performed using LUT-based multipliers rather than DSPs. As each PE requires

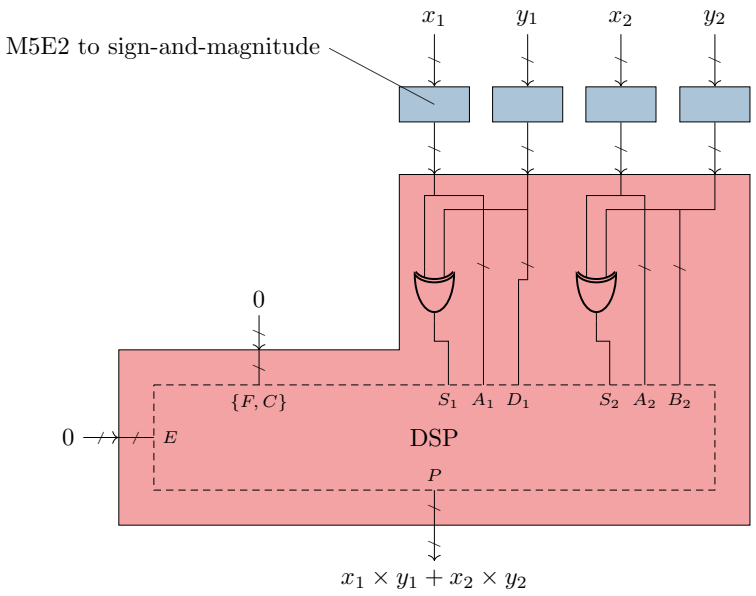

Figure 4.3: Two-element M5E2 dot product in full-precision with one DSP.

the computation of four 32-element dot products (as detailed in Figure 4.1), the module of Figure 4.3 must be replicated numerous times inside each PE. The simplest way to implement the 32-element dot product units needed in each PE is to chain the DSPs as shown in Figure 4.4. Note that, thanks to the F input of the DSPs, the LUT-based M5E2 multiplier can output their result in one's complement representation, which simplifies their design. In order to meet the resource constraints of the MPF300, four such LUT-based multipliers are required for each 32-element dot product.

This simple chain approach only uses a small numbers of LUTs because it only relies on the adders of the DSPs to accumulate the partial sums of the dot product. However, using

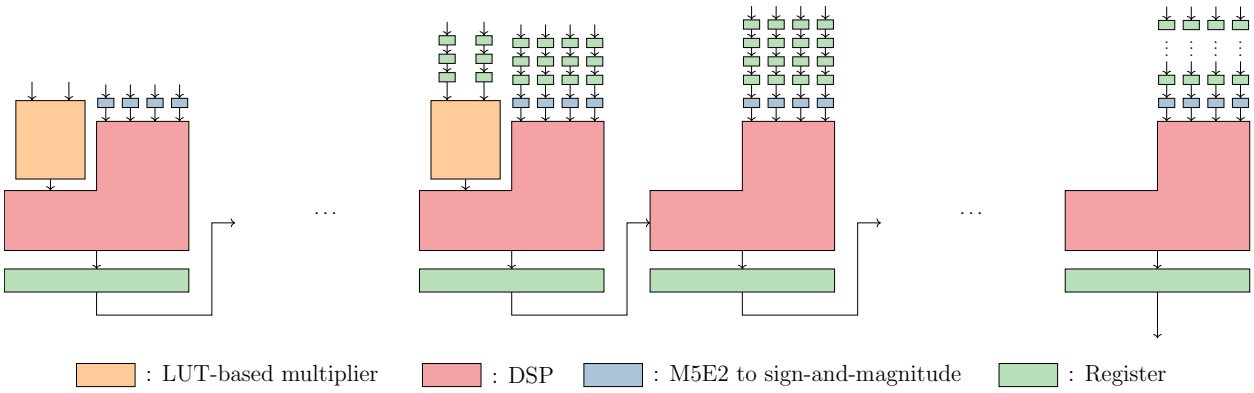

Figure 4.4: Dot product using a chain of DSPs.

this configuration, pipelining is highly impractical as the inputs of the final DSPs must be delayed by numerous clock cycles. This approach indeed requires at least 12000 flip-flops per PE. In total, for 16 PEs, this would correspond to more than 64% of the number of available flip-flops on the MPF300 FPGA. This would not leave enough resources for the rest of the OPU core, and therefore cannot be employed here.

To reduce the total number of registers needed, a tree-like arrangement of the DSPs is considered, as shown in Figure 4.5. Because it has 7 DSPs and 2 LUT-based multiplier, this circuit can perform a 16-element dot product. The PEs of the OPU core, which require 32-elements dot products, are implemented by using pairs of such modules and adding their result using a final LUT-based adder. Naturally, a three-level tree of DSPs could be used to implement the PEs directly, without any extra adder. However, it is found that that approach would still require a large number of flip-flops due to having more stages. In total,

using trees of 7 DSPs requires approximately 100× fewer flip-flops than when using chains of DSPs. In addition, using a tree allow a significant reduction in the total latency of the PEs.

Component	Count	LUTs	DSPs
LUT-based multiplier	16×	83	-
DSP-based multiplier	102×	-	0.5
M5E2 to fixed-point	128×	12	-
Other	4×	30	-
Total		1760	51

Table 4.5: Resources per PE when using trees of DSPs.

The resource utilization of this approach is detailed in Table 4.5. The number of LUTs required for this method is one order of magnitude smaller than when DSPs are solely used to perform mantissa multiplications. This is due to the fact that, with this tree-like approach, alignment and addition of the intermediate products can be done in the DSPs. Thanks to its efficient utilization of the DSPs, this strategy is the only one studied that can be used, in practice, to compute 2048 MAC operation per cycle on the MPF300 FPGA. As such, this PE design is the one that is implemented in the new OPU core presented in this thesis.

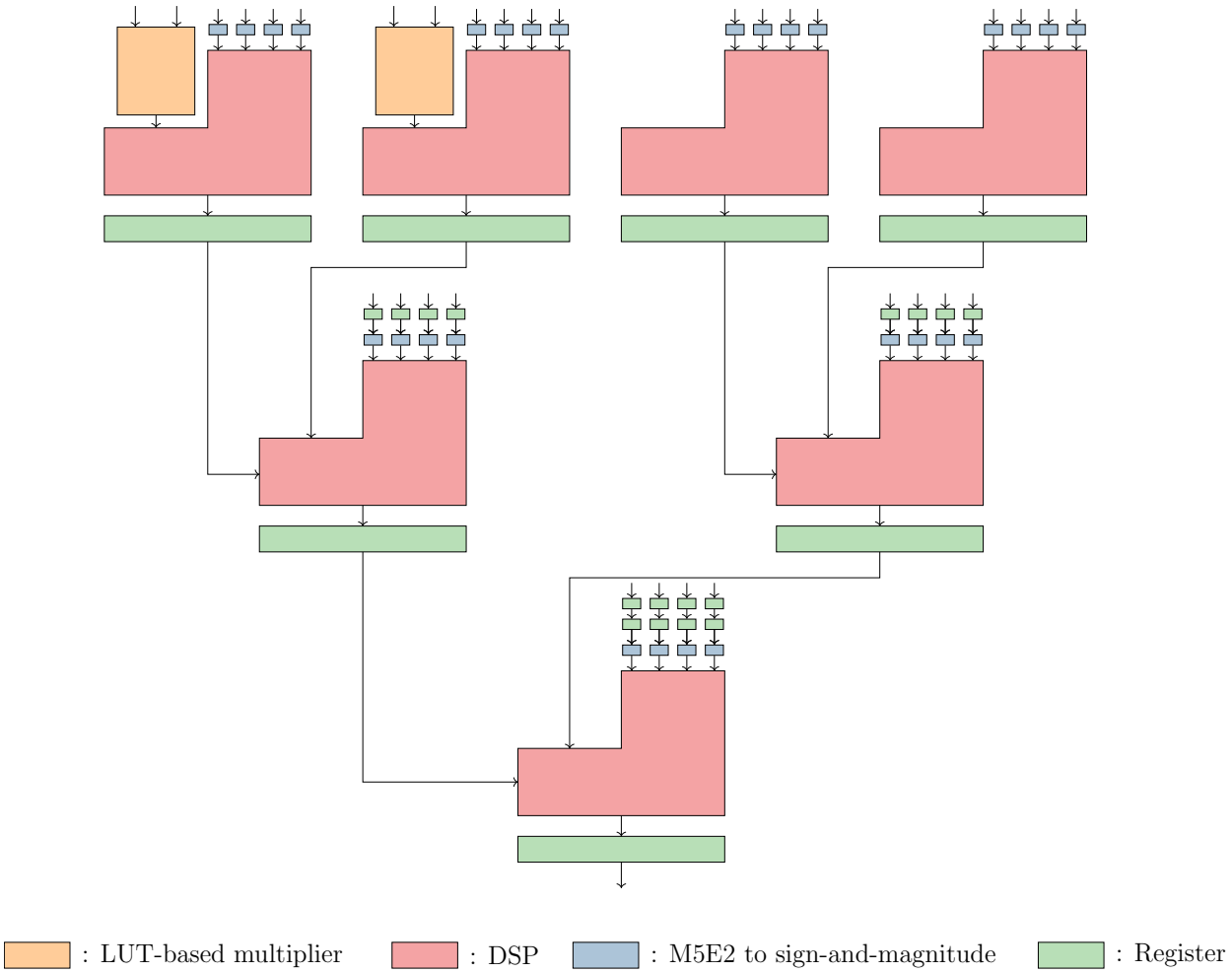

Figure 4.5: Dot product using a tree of DSPs.

Chapter 5

Core performance

This chapter details the performance characteristics of the new OPU core implemented on a MPF300 FPGA. In particular, this core is here compared to the two other existing implementations of the OPU ISA.

5.1 Resource utilization

Table 5.1 compares the resources required to implement the prior and new OPU cores. As mentioned in Chapter 3, the MPF300 device utilizes 4-input LUTs, whereas the Xilinx FPGAs contains slightly wider and more flexible LUTs. For a more adequate comparison, the number of equivalent *logic cells* is listed in Table 5.1 in addition to the number of LUTs. Here, it is considered that 1 Xilinx LUT is equivalent to 1.6 logic cells. This conversion factor is indeed the one employed by Xilinx Inc. for marketing purposes.

	OPU1024 [39]	OPU4096 [39]	New
Device	XC7K325T	XC7Z100	MPF300
Data format	INT8	INT8	M5E2
MAC/cycle	1024	4096	2048
LUT size	5 and 6	5 and 6	4
Logic cells	151621	247226	230375
LUTs	94763 (47%)	154516 (56%)	230375 (77%)
Flip-flops	150848 (37%)	337651 (61%)	254616 (85%)
DSPs	516 (61%)	1986 (98%)	898 (94%)

Table 5.1: Resource utilization of existing and proposed OPU cores.

For OPU implementations, it is reasonable to expect that the number of logic cells required grows linearly with `NUM_MAC`, the number of multiply-accumulate operations that can be performed per cycle. Based on the two existing cores, under this assumption, the expected number of logic cells needed for the new implementation is 183490. Yet, the actual LUT count is 230375, which is 25% larger than the prediction. This extra resource consumption can be mainly attributed to the adder support for 8-bit floating-point representation support.

Under the same linearity assumption, the expected number of used flip-flop for the new core is 213120. The actual flip-flop count is actually 19% larger, which is significant. This is attributed to the drastically different dot product unit design employed for this core in order to support floating-point arithmetic despite the low number of LUTs available. In particular, the implemented tree structure does employ a large number of flip-flops to delay

the operands of the last stages. In addition, for reasons explored in the following chapter, this design requires a large amount of resource duplication in order to meet the timing constraint. This also contributes significantly to the larger flip-flop utilization.

It is also worth noting that, because the different implementations of the OPU ISA have different numbers of DSP available on their respective FPGA devices, they must implement different fractions of the multiplications outside of the DSPs. The first core, OPU1024, can implement all of its MAC operations on DSPs. The second core, OPU4096, implements 97% of its MAC operations in DSPs. However, due to the relatively lower number of DSPs in the MPF300, only 87% of the MAC operations are performed in these in the new implementation. This has the effect of inflating the LUT utilization proportionally more than for the other two prior cores.

5.2 Performance & efficiency

In this section, the performance and efficiency of the new OPU implementation is analyzed. Table 5.2 compares this implementation with the two prior cores. One can notice that the three employed FPGAs are all manufactured using 28nm technologies, which makes their comparison relatively fair.

The major difference between the cores is the fact that original cores can both operate

a clock frequency of 200MHz, whereas the new implementation can only reach 130MHz. Because of that, this implementation's peak throughput is only 27% larger than the existing OPU1024 core, despite the fact that it can compute twice the number of operations per cycle. The new core is significantly slower than OPU4096, which is expected as the latter is mapped onto a much larger FPGA device.

	OPU1024 [39]	OPU4096 [39]	New
Device	XC7K325T	XC7Z100	MPF300
Technology node	28nm	28nm	28nm
LUT technology	SRAM	SRAM	NAND flash
Estimated average power (W)	6.4	?	3.8
Clock frequency (MHz)	200	200	130
MAC/cycle	1024	4096	2048
Peak throughput (GOP/s)	410	1638	532
Peak energy-efficiency (GOP/J)	64	?	140

Table 5.2: Theoretical performance of existing and proposed OPU core.

Table 5.2 reports the average power consumption of the FPGA chip for the different cores. A significant difference in consumption is observed between these cores. When running, the new core is indeed estimated to consume 40% less power than the similarly-sized OPU1024 implementation. Table 5.3 presents a breakdown of this power consumption per component type. The lower consumption is the result of two different factors. First, as explained in Chapter 3, the employed FPGA uses flash-based lookup tables, which are known to consume less power. Second, excluding I/Os, the core operates a significantly reduced frequency which

naturally reduces dynamic power consumption. Thanks to the reduced power consumption, the new implementation can provide a much higher theoretical maximum energy-efficiency compared to the similarly-sized OPU1024. With a 2.1× improvement in energy efficiency and 1.7× reduced power consumption, the newly implemented core has thus the potential to be even more advantageous for edge inference acceleration.

Component	Percentage
I/O	29.4%
Net	27.9%
Memory	14.0%
Gate	12.3%
DSP	12.2%
Core Static	3.3%

Table 5.3: Breakdown of power consumption by component type.

Naturally, unless the processing elements are all fully utilized at all time, the OPU core cannot reach its peak energy-efficiency. To evaluate this phenomenon, the MAC runtime efficiency (RME) of a design is here defined as the ratio of its obtained throughput for a given model to its theoretical peak throughput. As such, this metric measures the fraction of time that the processing elements are busy performing useful computations. The OPU compiler is designed to optimize the computation of neural network so that their RME is maximized. Thanks to this, OPU1024 can generally reach a RME between 85% to 98% in practice when accelerating real neural networks.

The same compiler is used is here used to evaluate the RME of the proposed OPU core. These realistic results are shown in Table 5.4 for three different models. For Resnet-50 [16], a popular neural network used for image classification, the RME of the compiled model on the new core is only slightly lower than when running on the narrower OPU1024 core. As a result, the increased number of multiply-accumulate operations performed per cycle is beneficial despite the lower clock frequency. A noticeably larger throughput is thus observed in practice. For the same reason, the new core provides a larger energy-efficiency than OPU1024. Similar results are obtained with YOLOv2 [33], a network designed for object detection. However, when considering the compact version of the model, Tiny-YOLO [32], the compiler is unable to find a scheduling that allows to use most of the 2048 MACs every cycle. In other words, the OPU core provide parallelism that Tiny-YOLO cannot exploit, and therefore, the effective RME is only of 53.5%. As a consequence, the new core yields a lower throughput in this situation. Nevertheless, one should note that thanks to the significantly reduced power consumption, the energy-efficiency of the new implementation is still 1.68× better than the existing OPU1024 core.

In average, the results of Table 5.4 thus show that the new OPU core consumes less power than OPU1024 while maintaining good performance, which makes it more energy-efficient. Because of this characteristics, this new core is generally superior to the existing OPU1024, especially for applications where power consumption and energy usage are constrained.

Core			Resnet-50	YOLOv2	Tiny-YOLO
OPU1024 [39]	1024 MAC/cycle at 200MHz	Frame/s	54.36	7.23	68.32
		RME	84.5%	95.5%	89.2%
		Efficiency (GOP/J)	56.5	61.1	57.2
New	2048 MAC/cycle at 130MHz	Frame/s	62.1	9.26	53.1
		RME	77.7%	94.2%	53.5%
		Efficiency (GOP/J)	108.	131.	96.3

Table 5.4: Runtime MAC efficiency (RME) of proposed OPU core.

As mentioned, the power consumption reported in Table 5.2 corresponds to the consumption of the FPGA device alone. In a realistic setting, however, the OPU core must be accompanied with other devices, including voltage regulators and DDR memory chips. For this reason, the OPU core is tested on an FPGA development board, which is measured to consume at most 9.8W when in operation, including power supply losses. The corresponding energy efficiency is reported in Table 5.5. In this table, the cores are also compared to the Nvidia Jetson Nano [9], an edge GPU with similar characteristics as the OPU cores.

Platform			Resnet-50	YOLOv2	Tiny-YOLO
OPU1024 [39]	410GOP/s 16.5W	RME	87.9%	95.5%	89.2%
		Efficiency (GOP/J)	21.9	23.7	22.2
New	532GOP/s 9.8W	RME	77.7%	94.2%	53.5%
		Efficiency (GOP/J)	42.1	51.1	29.0
Jetson Nano	472GOP/s 10W	Frame/s	36	4	25
		RME	50%	45%	28%
		Efficiency (GOP/J)	23.9	21.6	13.4

Table 5.5: Efficiency of the OPU core including system consumption.

When considering the power consumption of the entire system, the new implementation of the OPU ISA maintains its advantageous energy-efficiency. The new core is indeed approximately 2 times more efficient than OPU1024 when tested on Resnet-50 and YOLOv2, and 1.3 times more efficient when running Tiny-YOLO. As a general purpose GPU platform, because it is not optimized for this task, the Jetson Nano reaches significantly lower RME for all tested networks. As a result, despite its similar peak throughput, the effective throughput of this GPU is approximately two times lower than both OPU1024 and the new OPU core. While the existing OPU1024 core has a similar energy-efficiency as the Jetson Nano platform, the new implementation consumes, in average, half of the energy as this GPU when performing the same computation.

In summary, the new OPU core presented in this work is thus generally faster and consumes less power compared to both the Jetson Nano GPU and OPU1024. However, although the Nvidia Jeston Nano is here found to be a less energy-efficient platform than the new OPU implementation, it is worth noting that several more recent edge GPUs, such as the Nvidia Xavier NX, integrate dedicated arithmetic circuits to accelerate deep learning inference. Thanks to this, those newer GPUs typically outperform all FPGA-based implementations of OPU.

Chapter 6

Performance bottleneck

As detailed in the previous chapter, compared to the existing OPU1024 core, the new implementation is able to compute 2× more multiply-accumulate operations per cycle, but it operates at a clock frequency 1.5× lower. As a result, the peak throughput of this new core is larger than the existing core. However, experiments show that, while some networks such as YOLOv2 and Resnet-50 can make use of this extra parallelism, other small networks such as Tiny-YOLO cannot be mapped efficiently to a wider OPU core. Because of this, the new core is found to execute Tiny-YOLO more slowly than the existing one, despite its significantly higher peak throughput.

The existing OPU cores and the proposed one have a similar architecture, with similar functionality. As such, one would expect that they could operate a similar frequency, yet this is not the case. Naturally, the difference in FPGA technology could partially justify this

observed difference. However, when mapping each individual module separately onto the FPGA device, it is found that they all could run at a frequency of up to 400MHz. Mapping more than one module onto the FPGA causes the maximum reachable frequency to decrease, and to fall as low as 130MHz when considering the entire design. This clearly suggests that placement congestion is a major issue in the new OPU core, and that it is, most likely, the main factor limiting performance. In this chapter, factors that could potentially contribute to this problematic congestion are thus explored.

6.1 Floating-point support

As described in the previous chapter, the added support for 8-bit floating-point arithmetic represents extra complexity in the OPU core compared to exiting implementations. This has the potential to decrease the maximum reachable frequency in two different ways. First, a time critical path might be present in the floating-point dot product unit, while not being present in a similarly sized fixed-point unit. Second, as mentioned previously, a floating-point dot product unit has a non-negligible higher cost in LUTs and flip-flops than a fixed-point unit. As a result, floating-point support can lead to a greater utilization of routing resources, and cause congestion on the FPGA device.

While both mechanisms could possibly contribute to the observed congestion, evidence shows that neither can here justify the lower frequency of the design. Indeed, the datapaths

of the dot product unit are all found to have a significant positive timing slack in the final design, and therefore they could operate a much higher frequency than 130MHz. In addition, in order to test whether the added resource utilization of the floating-point support contributes significantly to the congestion, two different experiments are conducted: the core is mapped onto the FPGA with all timing constraints disabled inside the dot product unit, and the floating-point dot product unit is replaced by a simpler fixed-point variant. In both experiments, the design can only reach a frequency of 130MHz, just like the complete design. Given these results, it is reasonable to conclude that support for floating-point arithmetic does not noticeably impact the reachable clock frequency of the OPU core.

6.2 Dot product unit width

The existing OPU1024 core is implemented on a similarly-sized FPGA than the new OPU core. In addition to the floating-point support, the major difference between these two cores is the width of their dot product unit, i.e. the number of multiply-accumulate operations that they can compute per clock cycle. Because of its twice larger unit, the new core is relatively more complex, and it is therefore harder to generate efficient routing. To verify whether this added complexity is a significant factor contributing to the congestion, a new core for whose dot product unit width has been divided by two is mapped onto the FPGA. Despite being significantly smaller, this simpler design cannot reach a frequency higher than 135MHz. Though slightly faster than 130MHz, this obtained frequency, too, is significantly

lower than the 400MHz reached when congestion is artificially eliminated. This demonstrates than the added complexity of the new core is not, in itself, responsible for the observed low frequency.

6.3 Lookup table size

As described in Chapter 2, a major difference between the exiting OPU cores and the new one is that the latter is implemented on a FPGA that utilizes 4-input LUTs. By contrast, the existing cores are both implemented using wider and more flexible LUTs. Because they tend to reduce the number of logic stages, those wider LUTs could help increase the maximum reachable frequency. However, in both the existing and new OPU cores, the large majority of logic functions have an arity less than or equal to 4, and therefore they can be implemented using a single 4-input LUT. In addition, for most path delays, including the critical paths, the total gate delay is negligible compared to the net delay. The benefit of using wider LUTs for OPU is thus relatively small. It is thus believed that the smaller LUT size does not impact the clock frequency of the OPU core in this case.

6.4 Location of hard resources

The given observations seem to indicate that the intrinsic complexity of the OPU core is not the cause of the congestion and low clock frequency. Rather, the placement congestion

is mainly the consequence of the inadequate structure of the OPU core with respect to the FPGA device. Indeed, it is found that the location of the hard resources on the device does not match the general structure of the OPU core. This inadequacy then causes suboptimal logic placement and poor routing, which leads to a reduced reachable clock frequency.

As described in Chapter 3, the MPF300 FPGA contains three major types of hard resources: LSRAMs, μSRAMs, and DSPs. Figure 6.1 presents the high-level structure of the OPU core, along with the utilization of these hard resources in the core. The DSPs are naturally employed to implement the dot product unit. The LSRAM are used to create the *fm* and *ofm* buffers of the OPU architecture, while the μSRAM are used for the wider *ker* buffer. Approximately 50% of the available μSRAM blocks are utilized for the *ker* buffer, while only 10% of the LSRAMs are employed for the *fm* buffer. Close to 100% of the DSPs are utilized to implement the dot product unit.

Figure 6.2 shows the layout of the fabric of the MPF300 device. In this diagram, each represented *logic cluster* and *interface logic* contains a tight cluster of 12 logic cells. As clearly visible, the resources are arranged in rows of cells of the same type. Every four rows is dedicated for a specific type of hard resource, while the other rows contain only lookup tables and flip-flops. From top to bottom, the types of hard resource rows cycle according to the following order: DSP, LSRAM, then μSRAM. In this diagram, only part of the device is represented. In reality, the MPF300 has significantly more rows and columns than shown.

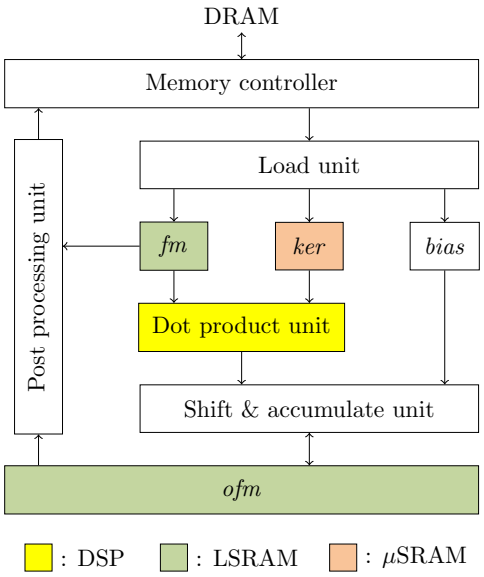

Figure 6.1: Utilization of hard resources in the OPU core.

A major problem of this layout is the fact that the DSPs are scattered across the entire device. Given that the dot product unit utilizes virtually all of these DSP blocks, this unit must necessarily be stretched across the entire device as well, as shown in Figure 6.3. As a result, all of the other components of the OPU core must be mapped in between the resources allocated to the dot product unit. Even though, without the placement constrains of hard resources, these components could be physically separated from each other, they must here compete for logic cells and routing resources. In turn, this causes the modules to be more stretched out than necessary, which lengthens their internal nets. Also, this competition for routing resources has the effect of creating less efficient routing due to congestion. Those two detrimental consequences of the distribution of DSPs on the MPF300

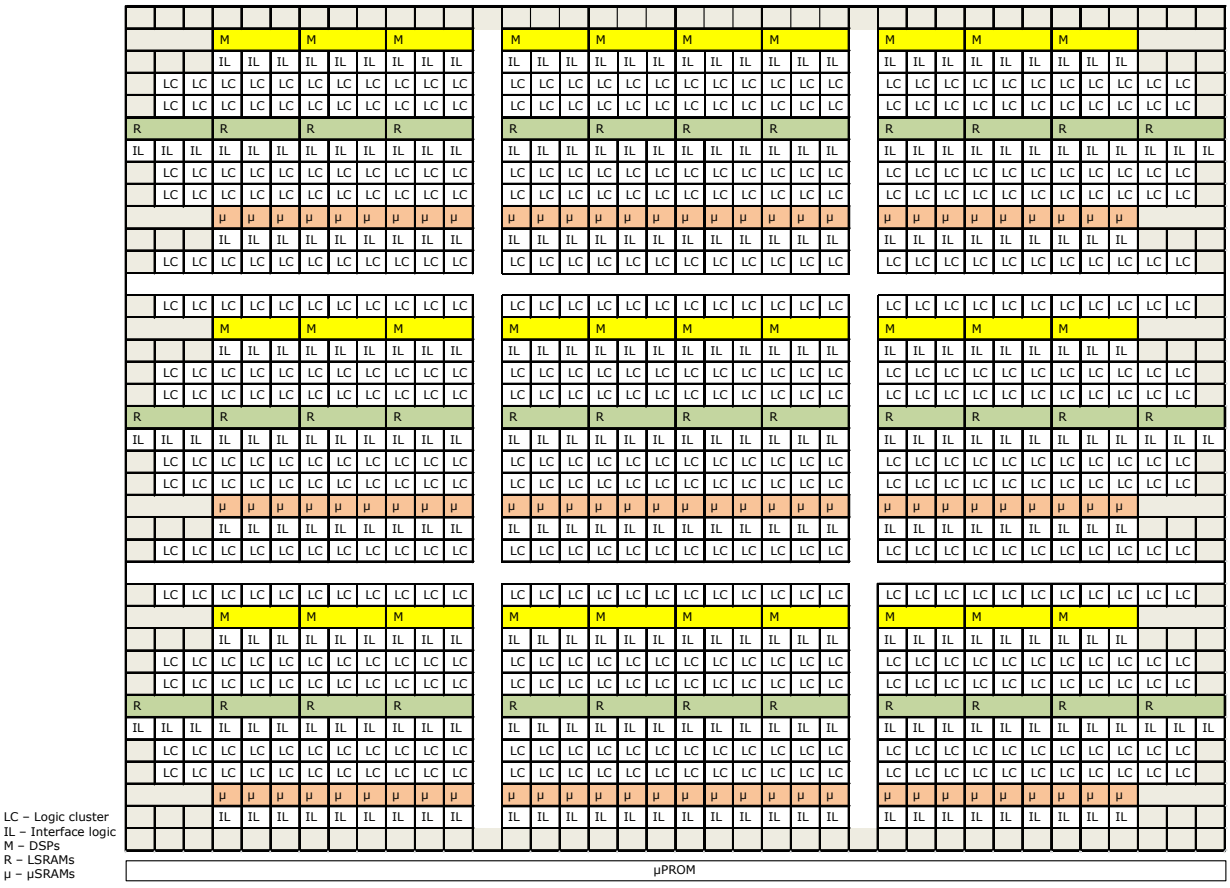

Figure 6.2: Global layout of the MFP300 FPGA fabric [31].

are believed to be the major causes of the low maximum frequency of the mapped OPU core.

Another cause of routing congestion are the buses that connects the *ker* buffer to the dot product unit. As visible in Figure 6.1, these buses consists of wires between μSRAM blocks and DSPs. However, in Figure 6.2, one can notice that these two resources are always separated by at least 3 rows of logic cells on the device. Given that roughly 100% of the

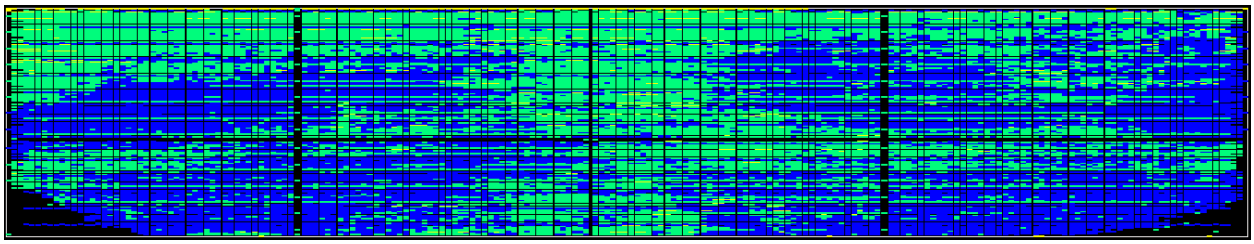

Figure 6.3: Placement of the dot product unit (in blue) on the FPGA.

DSPs are utilized and that all of them receive operands from the the *ker* buffer, this implies that most of the area of the FPGA is covered with wires that bring the outputs of μSRAMs to a nearby DSP. This represent an enormous utilization of routing resource, which hinders the ability to route efficiently all the other signals of the OPU core. As such, this factor also greatly contributes to the observed congestion and reduced frequency.

Possible improvements

The fabric of FPGAs such as the MPF300 is optimized for more traditional digital signal processing applications than machine learning. Indeed, as described, the hard resources are arranged in a manner that is most efficient when mapping multiple small modules that each require a few memory and DSP blocks. This arrangement is however not optimal when mapping large modules that require a large fraction of the hard resources of one specific type. Such a design would indeed map poorly to the FPGA because of the placement constraints imposed by these hard resources. In the OPU core, the large DSP utilization in the dot product unit is the most problematic. However, this issue is not restricted to OPU as

most FPGA-based accelerator for machine learning do also contain a large arithmetic unit that utilizes most of the DSP resources. In this context, and given the increasing popularity of deep learning, the development of FPGA devices specialized for machine learning applications would be highly valuable.

Different distribution of hard resources

Given the results found here for the OPU architecture, it seems that such a specialized FPGA would benefit from a different placement of the hard resources. In particular, these should not uniformly distributed on the device as they are in the MPF300. Rather, it would be preferable if regions with more concentrated hard resources were available. For instance, providing the same number of hard DSPs but on only one half of the device, as shown in Figure 6.4, would create a better separation of the different modules. As a consequence, competition for for logic and routing resources would be reduced throughout the entire device. As such, such a layout would, at least for OPU, permit an increased clock frequency. In addition, thanks to the better mapping, such an implementation would likely yield shorter nets in average, and therefore reduce the total dynamic power consumption.

More flexible memory resources

As previously mentioned, the relative location of the memory block and the DSPs is also a large source of wasteful routing utilization in the new core. In Xilinx FPGAs, this issue is

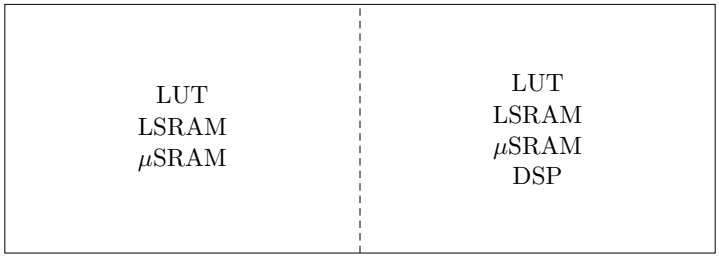

Figure 6.4: More adequate distribution of resources on the FPGA.

avoided thanks to the Distributed RAM blocks which can be instantiated almost anywhere on the device with very few constraints. However, such a flexible memory instantiation is only feasible on SRAM-based FPGAs, and not on flash-based ones such as the MPF300 employed here. Unfortunately, this means that a potential specialized FPGA could not benefit from both the higher power-efficiency of flash-based technology, and the improved routing of SRAM-based LUTs. Further research is however needed to properly quantify this trade-off.

Narrower DSP blocks

The structure of the DSPs available in current FPGAs is also suboptimal for machine learning. According to the finding described in Chapter 4, the best strategy to perform 8-bit floating-point dot products on the MPF300 is to convert the operands to full-precision fixed-point numbers then perform the dot product using DSPs. In that situation, because multiple bits of the operands are necessary null, the DSPs perform useless computations, which represents a waste a silicon area. Likewise, the DSPs are utilized suboptimally when using 8-bit

fixed-point arithmetic. Assuming that the size of a multiplier is proportional to the product of the number bits of its two operands, performing two 8×8 multiplications with the 19×18 multiplier of a DSP indeed corresponds to an actual DSP area utilization of

$$\frac{2 \times 8 \times 8}{19 \times 18} = 37.43\%^{1}.$$

Given this low utilization, it is clear that machine learning acceleration on FPGA would benefit from smaller DSPs that can only compute narrower products. This would allow to greatly increase the number of LUTs and DSPs that can be implemented on the same silicon area. Because of this, placement would be made easier and congestion would consequently be reduced, thereby increasing performance.

[1]Note that for Xilinx FPGAs, with 18×25 multipliers, this number is as low as 28.5%.

Chapter 7

Conclusion

The OPU ISA is a promising solution for the standardization of deep learning accelerators. As demonstrated by the performance and energy-efficiency of its two prior implementations, this ISA represents a valuable framework. In this work, a third implementation is designed for the MPF300 device. This new core is found to be, in average, 1.7× more energy-efficient than the existing OPU1024. Thanks to the flash technology of the employed FPGA, this core also provides added security and reliability. As such, this work contributes to the diversification of the available OPU cores.

Contribution

By comparing it with prior OPU cores, this work establishes that the MPF300 FPGA, too, is an adequate platform for the implementation of the OPU ISA. However, multiple obtained results contrast with prior findings regarding Xilinx FPGAs:

- 8-bit floating-point arithmetic is here found to be less area-efficient than 8-bit integer arithmetic.
- The more accurate M5E2 format is significantly more area-efficient than M4E3.
- The most area-efficient manner to perform large floating-point dot products is to convert the operands to wider integers, on the device, then complete the computations using integer arithmetic.
- Due to the lower ratio of the number of available LUTs to DSPs, the relative resource utilization is large and special consideration must be given in order to fit the entire core on the FPGA.

Future work

Thanks to the flash technology of the MPF300, the new core benefits from added reliability compared to the existing implementation of the OPU ISA. However, it is still vulnerable to potential soft errors. Further work is thus needed in order to develop an OPU core suitable for life-critical applications. In particular, a study of how the reliability of the core could be increased with limited impact on performance is necessary. Potential solutions include the combination of selective *triple module redundancy* and the exploitation of the intrinsic robustness of neural networks.

In this work, the placement of hard resources on the FPGA device, notably DSPs, contributes to congestion and decreases the performance of the OPU core. In that context, the development of modified FPGAs specialized for machine learning applications would be highly valuable, especially given the rapid evolution of the field of data science. Though a few recommendations are made in this work for such specialized FPGAs, further research is still required in order to determined their optimal layout. If well-designed, such an improved architecture could help FPGAs reach even greater performance and energy-efficiency, therefore increasing their relevance.

Bibliography

[1] Jeff Bar (Amazon.com). *Inf1 Instances with AWS Inferentia Chips for High Performance Cost-Effective Inferencing*. 2019. URL: https://aws.amazon.com/blogs/aws/amazon-ec2-update-inf1-instances-with-aws-inferentia-chips-for-high-performance-cost-effective-inferencing/ (visited on 05/01/2020).

[2] Krste Asanović and David A Patterson. "Instruction sets should be free: The case for risc-v". In: *EECS Department, University of California, Berkeley, Tech. Rep. UCB/EECS-2014-146* (2014).

[3] V. Badrinarayanan, A. Kendall, and R. Cipolla. "SegNet: A Deep Convolutional Encoder-Decoder Architecture for Image Segmentation". In: *IEEE Transactions on Pattern Analysis and Machine Intelligence* 39.12 (2017), pp. 2481–2495.

[4] Junjie Bai, Fang Lu, Ke Zhang, et al. *ONNX: Open Neural Network Exchange*. https://github.com/onnx/onnx. 2020.

[5] Tianshi Chen et al. “Diannao: A small-footprint high-throughput accelerator for ubiquitous machine-learning”. In: *ACM SIGARCH Computer Architecture News* 42.1 (2014), pp. 269–284.

[6] Y. Chen et al. “Eyeriss v2: A Flexible Accelerator for Emerging Deep Neural Networks on Mobile Devices”. In: *IEEE Journal on Emerging and Selected Topics in Circuits and Systems* 9.2 (2019), pp. 292–308.

[7] Sharan Chetlur et al. “cudnn: Efficient primitives for deep learning”. In: *arXiv preprint arXiv:1410.0759* (2014).

[8] Microsemi Corporation. *Low Power Leadership*. 2020. URL: `https://www.microsemi.com/product-directory/fpga-soc/1743-low-power` (visited on 04/29/2020).

[9] Nvidia Corporation. *Jetson Nano: Deep Learning Inference Benchmarks*. 2019. URL: `https://developer.nvidia.com/embedded/jetson-nano-dl-inference-benchmarks` (visited on 04/20/2020).

[10] Zidong Du et al. “ShiDianNao: Shifting vision processing closer to the sensor”. In: *Proceedings of the 42nd Annual International Symposium on Computer Architecture*. 2015, pp. 92–104.

[11] S. H. Fatemi Langroudi, T. Pandit, and D. Kudithipudi. “Deep Learning Inference on Embedded Devices: Fixed-Point vs Posit”. In: *2018 1st Workshop on Energy Efficient Machine Learning and Cognitive Computing for Embedded Applications (EMC2)*. 2018, pp. 19–23.

[12] Leon A. Gatys, Alexander S. Ecker, and Matthias Bethge. “Image Style Transfer Using Convolutional Neural Networks”. In: *The IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*. June 2016.

[13] Yunchao Gong et al. *Compressing Deep Convolutional Networks using Vector Quantization*. 2014. arXiv: `1412.6115 [cs.CV]`.

[14] Song Han, Huizi Mao, and William J Dally. “Deep compression: Compressing deep neural networks with pruning, trained quantization and huffman coding”. In: *arXiv preprint arXiv:1510.00149* (2015).

[15] Mohammad Havaei et al. “Brain tumor segmentation with Deep Neural Networks”. In: *Medical Image Analysis* 35 (2017), pp. 18–31.

[16] Kaiming He et al. “Deep Residual Learning for Image Recognition”. In: *The IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*. June 2016.

[17] Yihui He, Xiangyu Zhang, and Jian Sun. “Channel Pruning for Accelerating Very Deep Neural Networks”. In: *The IEEE International Conference on Computer Vision (ICCV)*. Oct. 2017.

[18] S. Heller et al. “Hardware Implementation of a Performance and Energy-optimized Convolutional Neural Network for Seizure Detection”. In: *2018 40th Annual International Conference of the IEEE Engineering in Medicine and Biology Society (EMBC)*. 2018, pp. 2268–2271.

[19] "IEEE Standard for Floating-Point Arithmetic". In: *IEEE Std 754-2008* (2008), pp. 1–70.

[20] Satoshi Iizuka, Edgar Simo-Serra, and Hiroshi Ishikawa. "Let There Be Color! Joint End-to-End Learning of Global and Local Image Priors for Automatic Image Colorization with Simultaneous Classification". In: *ACM Trans. Graph.* 35.4 (July 2016). ISSN: 0730-0301.

[21] Benoit Jacob et al. "Quantization and training of neural networks for efficient integer-arithmetic-only inference". In: *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*. 2018, pp. 2704–2713.

[22] Jeff Johnson. *Rethinking floating point for deep learning*. 2018. arXiv: 1811.01721 [cs.NA].

[23] Norman P. Jouppi et al. *In-Datacenter Performance Analysis of a Tensor Processing Unit*. 2017. arXiv: 1704.04760 [cs.AR].

[24] Dhiraj Kalamkar et al. *A Study of BFLOAT16 for Deep Learning Training*. 2019. arXiv: 1905.12322 [cs.LG].

[25] Alex Krizhevsky, Ilya Sutskever, and Geoffrey E Hinton. "ImageNet Classification with Deep Convolutional Neural Networks". In: *Advances in Neural Information Processing Systems 25*. Ed. by F. Pereira et al. Curran Associates, Inc., 2012, pp. 1097–1105.

[26] Liangzhen Lai, Naveen Suda, and Vikas Chandra. "Deep convolutional neural network inference with floating-point weights and fixed-point activations". In: *arXiv preprint arXiv:1703.03073* (2017).

[27] Hao Li et al. "Training Quantized Nets: A Deeper Understanding". In: *Advances in Neural Information Processing Systems 30*. Ed. by I. Guyon et al. Curran Associates, Inc., 2017, pp. 5811–5821.

[28] X. Lian et al. "High-Performance FPGA-Based CNN Accelerator With Block-Floating-Point Arithmetic". In: *IEEE Transactions on Very Large Scale Integration (VLSI) Systems* 27.8 (2019), pp. 1874–1885.

[29] Jian-Hao Luo, Jianxin Wu, and Weiyao Lin. "ThiNet: A Filter Level Pruning Method for Deep Neural Network Compression". In: *The IEEE International Conference on Computer Vision (ICCV)*. Oct. 2017.

[30] Martín Abadi et al. *TensorFlow: Large-Scale Machine Learning on Heterogeneous Systems*. Software available from tensorflow.org. 2015. URL: `https://www.tensorflow.org/`.

[31] Microsemi Corporation. *UG0680 - User Guide - PolarFire FPGA Fabric*. Microsemi Corporation. Aliso Viejo, California, U.S.A., 2020.

[32] Joseph Redmon and Ali Farhadi. *YOLOv3: An Incremental Improvement*. 2018. arXiv: `1804.02767 [cs.CV]`.

[33] Joseph Redmon et al. “You Only Look Once: Unified, Real-Time Object Detection”. In: *The IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*. June 2016.

[34] M. Roukhami et al. “Very Low Power Neural Network FPGA Accelerators for Tag-Less Remote Person Identification Using Capacitive Sensors”. In: *IEEE Access* 7 (2019), pp. 102217–102231.

[35] Chen Wu et al. *Low Precision Floating-point Arithmetic for High Performance FPGA-based CNN Acceleration*. 2020. arXiv: 2003.03852 [eess.SP].

[36] Ephrem Wu et al. “Compute-Efficient Neural-Network Acceleration”. In: *Proceedings of the 2019 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays*. FPGA ’19. Seaside, CA, USA: Association for Computing Machinery, 2019, pp. 191–200.

[37] Xilinx Inc. *UG473 - User Guide - 7 Series FPGAs Memory Resources*. 2019.

[38] Xilinx Inc. *UG479 - User Guide - 7 Series DSP48E1 Slice*. 2019.

[39] Y. Yu et al. “OPU: An FPGA-Based Overlay Processor for Convolutional Neural Networks”. In: *IEEE Transactions on Very Large Scale Integration (VLSI) Systems* 28.1 (2020), pp. 35–47.